Haskell Library for Quantitative Analysis in Finance

Midway Report

Student:	Kasper Passov
Supervisors:	Fritz Henglein
	Jost Berthold

Contents

1 Problem Description

2 Problem Definition

I want to design and develop a Haskell library for quantitative finance, taking the open source `QuantLib` as a starting point. The language Haskell is used because of its purity and advanced type class system which allows it to model relevant entities.
The result of my project should be a software architecture that supports different kinds of financial instruments and valuation methods.

3 Learning Goals

The following are the goals i hope to fulfill during this project:

- I will be able to work with models of different financial instruments and pricing methods.
- I will be able to structure and execute a medium-sized software project in a functional language.
- I will be able to implement parts of a medium-sized financial project using Haskell.
- I will be able to use advanced Haskell typesystem features.

Overall i hope to gain a deeper knowledge of Haskell and functional programming, an overview of the basic finacial instruments and methods and gain experience in structuring and completing a sizeable project.

4 Project Specifications

The project will be an extension of the Haskell library `HQL`, implementing some of the functionality of the `C++` library `QuantLib`[2], onto the current system.
The `QuantLib` library is writen in object oriented programming style (`C++`) and uses a class system, which is not strictly translateable into the Haskell type class system. Haskell type classes ensures that some operations will be availabe for values of chosen types, meaning they are more similar to abstract classes with multiple inheritance, such as the `Java Interface`, than with the actual `C++` classes.
This raises the problem of a class architecture that cannot be directly translatet, as a `C++` class cannot translate into a Haskell type class. I will analyse and extract the architecture of `QuantLib` and redesign it to accomodate the new type class system, without removing functionality.
The main milestone of the project is to implement the vanilla swap functionality into the current `HQL` library. This will have several hurdles, as a vanilla swap

is a swap of a fixed interest rate with a floating rate. At the time of writing, floating interest rate is not implemented in HQL, so another part of the project would entail the addition of this functionality.

5 Project Limitations

As QuantLib is a large library, implementing everything would be a much to large task for a bachelor project, meaning only parts of the QuantLib functionality will be added to HQL.
As statet above, the main milestone of the project is the implementation of vanilla swaps, and all dependant functionality. After the milestone is reached, further analysis of QuantLib will be done, resulting in selection and prioritisation of more expansions to the HQL library. I will however ignore date calculations in my implementation as this broadens the scope beyond what is possible of this project.

6 Project Justification

6.1 Justification of a Quantitative Library

The field of Quantitative analysis is calling for evermore computational power, to do the calculations needed for the exponetialy expanding data volumes, using the computation-intesive methods for complex risk analysis.
The context my project will take base on is the quantitative library HQL[3], developed within the HIPERFIT[6] research center, by master students Andreas Bock and Johan Astborg supervised by Jost Berthold and Sinan Gabel. The library is one in a chain of projects made by HIPERFIT, striving to develop functional programming solutions designed for highly parallel computation architectures. Making it easier to implement massive parallelization into (among others) financial methods and instruments[7].

6.2 Language Justification

Andreas Bock and Johan Astborg was given the language Haskell for development of HQL[3], (as part of HIPERFITs general goal of creating financialy highlevel functional programing tools[6]). As my project is based on HQL, the justification of the language is largely the same, and i will paraphase their reasoning.
The correctness of the program is paramount in the development of a quantitative library, as the finance sector is very suspectible to faulty software, where even small errors can lead to large economical damage. Because of this, we wish our language capable of catching as many errors as possible in the compilation phase.
In this Haskell excels, as its strict type system allows us to eliminate any type

errors during compilation whereas `C++`, some of these errors might first be found in runtime. Furthermore Haskell has a number of attractive features making it suited for financial calculations, including modularity[4], laziness[5], functionality, purity and type classes.
The purity makes it easier to translate equations into code because of maths inherent purity, and as different financial products often shares similar functionality, the functional programming of Haskell enables us to re-use code or create new functionality from existing one. Haskell allows overloading functionality based on types using type classes, this is extremely desirable as it allows us to have the same API regardless of exact type (e.g. an `Instrument` can be used abstractly as a parameter, so functions can have different functionality based on the exact instance of `Instrument`).

7 Problem Analysis

7.1 Financial Concepts

Things to explain.
Cashflow Coupons, Fixed, LIBOR, basic points, maturity, Instrument, Swaps, VanillaSwap

8 Architecture Design

8.1 Analysis

Detailing previous analysis of c++ architecture.

8.2 Implementation

Theory behind Model/Instrument/Pricing Engine, and remodeling of existing HQL splitting the Model and Instrument into seperat entities.
Justification of class hirachies (split them so future additions such as Commodities and Dividents can be added onto the system easily).

8.2.1 Cashflow

Like in QuantLib i would like to create a hierachy of classes where the most general class holds as much functionality as possible. For the legs of our Swap, we would like to have a floating leg and a fixed leg. These can be described as Coupons, or even more generaly, as Cashflows. In QuantLib the Cashflow base interface contains among others, a method to return the amount of a Cashflow. The Quantlib Cashflow class also contains methods regarding the dates of the Cashflow, and a hook, this is not modeled in HQL as of this work.

```
class Cashflow a where
    amount :: a -> Double
```

8.2.2 Coupons

Inherents from Cashflow. accrued rate, returning the interest rate accrued by the coupon. dayCounter, daycount convention. accrued ammount, takes a date and returns the accrued amount of cash from the Coupon until the given date. Wrap these in a monad or disregard them?

```
class Cashflow a => Coupon a where
    rate :: a -> Rate
```

8.2.3 Fixed and Floating Rate Coupon

Instances of Coupons. More work in swap to find out exacly what methods are needed.

```
data FixedRateCoupon = FixedRateCoupon Rate -- InterestRate?
instance Coupon FixedRateCoupon where
  rate (FixedRateCoupon r) = r
```

Something interesting on the implementation. Possibly the usage of typefamilies to allow Continuously and Simple Rate with little extra code.

8.2.4 Swaps

More hirachy and inheretence. Description of the chosen functionality and implementation of it.

```
class Instrument s => Swap s where
  -- | returns whenever the swap is expired
  isExpired :: s -> bool
  -- | startdate of the swap
  startDate :: s -> Date
  -- | calculates the maturity date of the swap
  maturityDate :: s -> Date
  -- | calculates the net present value of the swap
  legNPV :: s -> Double
  -- | calculates the basis point of the swap
  legBPS :: s -> Double
```

Maybe use a multiparameterclass here

8.2.5 VanillaSwaps

Description of the chosen functionality and implementation of it.

```
data VanillaSwap = VanillaSwap FixedRateCoupon FloatingRateCoupon
```

References

[1] Bernd Bruegge, Allen H.Dutoit *Object-Oriented Software Engineering Using UML, Patterns and Java*, Person New International Edition, Third Edition, 2014.

[2] Luigi Ballabio *Implementing `QuantLib`*, Draft, 2013.

[3] Andreas Bock, johan Astborg, Jost Berthold, Sinan Gabel *HIPERFIT Quant Library*, 2014.

[4] John Hughes, The University Glasgow *Why Functional Programming Matters* 1990.

[5] Simon Peyton Jones, Jean-Marc Eber, Julian Seward *Composing contracts: an adventure in financial engineering* 2000.

[6] http://hiperfit.dk/

[7] Jost Berthold, Andrzej Filinski, Fritz Henglein, Ken Friis Larsen, Mogens Steffensen, Brian Vinter *Functional High Performance Financial IT* 2012.